# SMART CONTRACT AUDIT REPORT

for

# FWX LBPs

Prepared By: Xiaomi Huang

PeckShield

January 27, 2023

## Document Properties

| | |
|---|---|
| Client | Forward Enterprise Limited |
| Title | Smart Contract Audit Report |
| Target | FWX LBPs |
| Version | 1.0 |
| Author | Stephen Bie |
| Auditors | Stephen Bie, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | January 27, 2023 | Stephen Bie | Final Release |
| 1.0-rc | January 17, 2023 | Stephen Bie | Release Candidate |

## Contact

For more information about this document and its contents, please contact PeckShield Inc.



| | |
|---|---|
| Name | Xiaomi Huang |
| Phone | +86 183 5897 7782 |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the design document and related smart contract source code of the FWX LBPs protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About FWX LBPs

FWX LBPs (i.e., FWX Lending and Borrowing Pools) is a decentralized non-custodial liquidity markets protocol, which provides lending and borrowing services. The FWX LBPs protocol allows users to participate as depositors or borrowers. Depositors provide liquidity to the market to earn passive income, while borrowers are able to borrow in an over-collateralized fashion. Additionally, the depositors can also earn FWX, which is the governance token of FWX ecosystem. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of FWX LBPs

| Item | Description |
|---|---|
| Target | FWX LBPs |
| Type | EVM Smart Contract |
| Language | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | January 27, 2023 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/Forward-Development/Forward-Defi-Protocol-LBPs.git (4ba28d2)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Forward-Development/Forward-Defi-Protocol-LBPs.git (35dff6f)

## 1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

| Impact \ Likelihood | *High* | *Medium* | *Low* |
|---|---|---|---|
| *High* | Critical | High | Medium |
| *Medium* | High | Medium | Low |
| *Low* | Medium | Low | Low |

**Likelihood**



## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

| Category | Check Item |
|---|---|
| Basic Coding Bugs | Constructor Mismatch |
| | Ownership Takeover |
| | Redundant Fallback Function |
| | Overflows & Underflows |
| | Reentrancy |
| | Money-Giving Bug |
| | Blackhole |
| | Unauthorized Self-Destruct |
| | Revert DoS |
| | Unchecked External Call |
| | Gasless Send |
| | Send Instead Of Transfer |
| | Costly Loop |
| | (Unsafe) Use Of Untrusted Libraries |
| | (Unsafe) Use Of Predictable Variables |
| | Transaction Ordering Dependence |
| | Deprecated Uses |
| Semantic Consistency Checks | Semantic Consistency Checks |
| Advanced DeFi Scrutiny | Business Logics Review |
| | Functionality Checks |
| | Authentication Management |
| | Access Control & Authorization |
| | Oracle Security |
| | Digital Asset Escrow |
| | Kill-Switch Mechanism |
| | Operation Trails & Event Generation |
| | ERC20 Idiosyncrasies Handling |
| | Frontend-Contract Integration |
| | Deployment Consistency |
| | Holistic Risk Management |
| Additional Recommendations | Avoiding Use of Variadic Byte Array |
| | Using Fixed Compiler Version |
| | Making Visibility Level Explicit |
| | Making Type Inference Explicit |
| | Adhering To Function Declaration Strictly |
| | Following Other Best Practices |

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

| Category | Summary |
|---|---|
| **Configuration** | Weaknesses in this category are typically introduced during the configuration of the software. |
| **Data Processing Issues** | Weaknesses in this category are typically found in functionality that processes data. |
| **Numeric Errors** | Weaknesses in this category are related to improper calculation or conversion of numbers. |
| **Security Features** | Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.) |
| **Time and State** | Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads. |
| **Error Conditions, Return Values, Status Codes** | Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. |
| **Resource Management** | Weaknesses in this category are related to improper management of system resources. |
| **Behavioral Issues** | Weaknesses in this category are related to unexpected behaviors from code that an application uses. |
| **Business Logics** | Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application. |
| **Initialization and Cleanup** | Weaknesses in this category occur in behaviors that are used for initialization and breakdown. |
| **Arguments and Parameters** | Weaknesses in this category are related to improper use of arguments or parameters within function calls. |
| **Expression Issues** | Weaknesses in this category are related to incorrectly written expressions within code. |
| **Coding Practices** | Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained. |

# 2 Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the FWX LBPs implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 1 | ■ |
| High | 0 | |
| Medium | 1 | ■ |
| Low | 3 | ■ ■ ■ |
| Informational | 1 | ■ |
| Undetermined | 1 | ■ |
| Total | 7 | |

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 critical-severity vulnerability, 1 medium-severity vulnerability, 3 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key FWX LBPs Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Critical | Same Borrow Asset Enforcement in CoreBorrowing::_borrow() | Business Logic | Fixed |
| PVE-002 | Low | Token Decimal Normalization in APHCore::addLossInUSD() | Business Logic | Fixed |
| PVE-003 | Low | Revisited Logic of PoolBaseFunc::_getNextLendingForwInterest() | Business Logic | Fixed |
| PVE-004 | Low | Incompatibility with Deflationary/Rebasing Tokens | Business Logic | Confirmed |
| PVE-005 | Informational | Immutable States If Only Set at Constructor() | Coding Practices | Fixed |
| PVE-006 | Medium | Trust Issue of Admin Keys | Security Features | Mitigated |
| PVE-007 | Undetermined | Potential Protocol Risk from Low-Liquidity Assets | Business Logic | Confirmed |

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

# 3 | Detailed Results

## 3.1 Same Borrow Asset Enforcement in CoreBorrowing::_borrow()

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: CoreBorrowing
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

In the FWX LBPs protocol, the CoreBorrowing contract is one of the main entries. It allows the user to borrow assets in an over-collateralized fashion. In particular, the borrow() routine is designed to meet the requirement. While examining the logic of _borrow() called inside the borrow() routine, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the CoreBorrowing contract. By design, the user is identified by the Membership NFT in the protocol. Inside the _borrow() routine, when the user borrows assets with the supported collateral assets, a new loan position will be created for him to record the borrowTokenAddress, borrowAmount, collateralTokenAddress, and collateralAmount. Especially, if the user intends to reuse the previous loan position to borrow again, the borrowTokenAddress and collateralTokenAddress should keep consistent with the previous position. However, it comes to our attention that only the collateralTokenAddress is verified (line 260) inside the _borrow() routine. That is to say, a malicious actor can borrow different assets by using the same loan position, which directly undermines the design. Given this, we suggest to add necessary sanity check as below: require(loan.borrowTokenAddress == borrowTokenAddress) (line 260).

```
204     function _borrow(
205         uint256 loanId,
206         uint256 nftId,
207         uint256 borrowAmount,
```

```
        address borrowTokenAddress,
        uint256 collateralSentAmount,
        address collateralTokenAddress,
        uint256 newOwedPerDay,
        uint256 interestRate
    ) internal returns (Loan memory) {
        require(
            msg.sender == assetToPool[borrowTokenAddress],
            "CoreBorrowing/permission-denied-for-borrow"
        );

        require(
            assetToPool[collateralTokenAddress] != address(0),
            "CoreBorrowing/collateral-token-address-is-not-allowed"
        );

        Loan storage loan;
        LoanExt storage loanExt;

        // [newLoan, owedPerDay, maxLTV, rate, precision]
        uint256[] memory numberArray = new uint256[](5);

        PoolStat storage poolStat = poolStats[msg.sender];
        poolStat.updatedTimestamp = uint64(block.timestamp);
        poolStat.totalBorrowAmount += borrowAmount;

        if (loanId == 0) {
            currentLoanIndex[nftId] += 1;
            loanId = currentLoanIndex[nftId];
            numberArray[0] = 1;
        } else {}

        loan = loans[nftId][loanId];
        loanExt = loanExts[nftId][loanId];

        if (numberArray[0] == 1) {
            // Setup new loans
            loan.borrowTokenAddress = borrowTokenAddress;
            loan.collateralTokenAddress = collateralTokenAddress;
            loan.owedPerDay = newOwedPerDay;
            loan.lastSettleTimestamp = uint64(block.timestamp);

            loanExt.initialBorrowTokenPrice = _queryRateUSD(borrowTokenAddress);
            loanExt.initialCollateralTokenPrice = _queryRateUSD(collateralTokenAddress);
            loanExt.active = true;
            loanExt.startTimestamp = uint64(block.timestamp);

            poolStat.borrowInterestOwedPerDay += newOwedPerDay;
        } else {
            // Update existing loan
            require(loanExt.active == true, "CoreBorrowing/loan-is-closed");

```

```
            require(
                loan.collateralTokenAddress == collateralTokenAddress,
                "CoreBorrowing/collateral-token-not-matched"
            );

            _settleBorrowInterest(loan);

            // Rollover loan if it is overdue.
            if (loan.rolloverTimestamp < block.timestamp) {
                _rollover(loanId, nftId, msg.sender);
            }

            numberArray[1] = loan.owedPerDay;
            // owedPerDay = [(r1/365 * (ld-now) * p1) + (r2/365 * ld * p2) + (r2/365 * (
                leftover) * p1)] / ld
            loan.owedPerDay =
                ((loan.owedPerDay * (loan.rolloverTimestamp - block.timestamp)) +
                    (newOwedPerDay * loanDuration) +
                    ((interestRate *
                        loan.borrowAmount *
                        (loanDuration - ((loan.rolloverTimestamp - block.timestamp)))) /
                        (365 * WEI_PERCENT_UNIT))) /
                loanDuration;

            poolStat.borrowInterestOwedPerDay =
                poolStat.borrowInterestOwedPerDay +
                loan.owedPerDay -
                numberArray[1];
        }

        ...
    }
```

Listing 3.1: `CoreBorrowing::_borrow()`

**Recommendation** Revisit the implementation of the above `_borrow()` routine to enforce the borrow asset remains the same.

**Status** The issue has been addressed by the following commit: `7a34fa2`.

## 3.2 Token Decimal Normalization in APHCore::addLossInUSD()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `APHCore`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

In the `FWX LBPs` protocol, the user is identified by the `Membership NFT` and the bad debt of the lending pool is shared by all the depositors of the pool. The `APHCore::addLossInUSD()` routine is designed to accumulate the total loss of the user (specified by the input `nftId`, i.e., `Membership NFT`) in all lending pools. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `APHCore` contract. Inside the `addLossInUSD()` routine, the statement of `lossAmount = lossAmount * _queryRateUSD(IAPHPool(msg.sender).tokenAddress())` (line 91) is executed to estimate the loss with `USD`. However, it ignores the fact that the `decimals` of the tokens supported in different lending pools may be different. Given this, we suggest to normalize the different token `decimals` to the same.

```
88     function addLossInUSD(uint256 nftId, uint256 lossAmount) external {
89         require(poolToAsset[msg.sender] != address(0), "APHCore/caller-is-not-pool");
90
91         lossAmount = lossAmount * _queryRateUSD(IAPHPool(msg.sender).tokenAddress());
92         nftsLossInUSD[nftId] = nftsLossInUSD[nftId] + lossAmount;
93         totalLossInUSD = totalLossInUSD + lossAmount;
94
95         emit AddLossInUSD(address(this), msg.sender, nftId, lossAmount);
96     }
```

Listing 3.2: `APHCore::addLossInUSD()`

**Recommendation** Improve the implementation of the `addLossInUSD()` routine as above-mentioned.

**Status** The issue has been addressed by the following commit: `54123d0`. Especially, the team has confirmed that all the supported tokens in the lending pools have the same `decimals` (i.e., `18`).

## 3.3 Revisited Logic of PoolBaseFunc::_getNextLendingForwInterest()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PoolBaseFunc`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

In the `FWX LBPs` protocol, the `PoolBaseFunc` contract is designed to calculate the borrowing and lending interest rate. In particular, the `_getNextLendingForwInterest()` routine is designed to calculate the `FWX` token interest rate for the depositors. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `PoolBaseFunc` contract. Inside the `_getNextLendingForwInterest()` routine, the formula of `(IAPHCore(coreAddress).forwDisPerBlock(address(this))* (365 days / BLOCK_TIME)* rate)/ precision` converts the `FWX` token amount that will be allocated to the lending pool in one year into the amount of the `tokenAddress` specified token. According to the formula, we believe the calculation of `rate` and `precision` (line 74) is incorrect. It should be revised as below: `(uint256 rate, uint256 precision)= IPriceFeed(IAPHCore(coreAddress).priceFeedAddress()).queryRate(forwAddress, tokenAddress)` (line 74).

Moreover, we notice the precision of the `interestRate` is `WEI_UNIT` (i.e., `10**18`). After further analysis, we believe it should be `WEI_PERCENT_UNIT` (i.e.,`10**20`).

```
69      function _getNextLendingForwInterest(uint256 newDepositAmount)
70          internal
71          view
72          returns (uint256 interestRate)
73      {
74          (uint256 rate, uint256 precision) = IPriceFeed(IAPHCore(coreAddress).
                priceFeedAddress())
75              .queryRate(tokenAddress, forwAddress);
76
77          uint256 ifpPrice = _getInterestForwPrice();
78
79          uint256 newIfpTokenSupply = ifpTokenTotalSupply +
80              ((newDepositAmount * WEI_UNIT) / ifpPrice);
81
82          if (newIfpTokenSupply == 0) {
83              interestRate = 0;
84          } else {
85              interestRate =
86                  (IAPHCore(coreAddress).forwDisPerBlock(address(this)) *
```

```
                    (365 days / BLOCK_TIME) *
                    rate *
                    WEI_UNIT) /
                (newIfpTokenSupply * precision);
        }
    }
```

Listing 3.3: `PoolBaseFunc::_getNextLendingForwInterest()`

**Recommendation** Revisit the implementation of the above `_getNextLendingForwInterest()` routine.

**Status** The issue has been addressed by the following commit: `1225e67`.

## 3.4 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PoolLending/PoolBorrowing`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

In the `FWX LBPs` protocol, the `PoolLending` contract is one of the main entries for interaction with users. In particular, one entry routine, i.e., `deposit()`, accepts the deposits of the supported `tokenAddress` token. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(uint256 nftId, uint256 depositAmount)
        external
        payable
        nonReentrant
        whenFuncNotPaused(msg.sig)
        settleForwInterest
        returns (
            uint256 mintedP,
            uint256 mintedAtp,
            uint256 mintedItp,
            uint256 mintedIfp
        )
    {
        require(
            tokenAddress == wethAddress  msg.value == 0,
```

```
            "PoolLending/no-support-transfering-ether-in"
        );
        nftId = _getUsableToken(msg.sender, nftId);

        if (tokenHolders[nftId].pToken != 0) {
            require(lenders[nftId].rank == _getNFTRank(nftId), "PoolBaseFunc/nft-rank-
                not-match");
        } else {
            lenders[nftId].rank = _getNFTRank(nftId);
            lenders[nftId].updatedTimestamp = uint64(block.timestamp);
        }

        _transferFromIn(msg.sender, address(this), tokenAddress, depositAmount);
        (mintedP, mintedAtp, mintedItp, mintedIfp) = _deposit(msg.sender, nftId,
            depositAmount);
    }
```

Listing 3.4: `PoolLending::deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into the `FWX LBPs` protocol. In the `FWX LBPs` protocol, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

**Recommendation** If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

**Status** The issue has been confirmed by the team. There is no need to support deflationary/rebasing tokens.

## 3.5 Immutable States If Only Set at Constructor()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Vault/HelperBase
- Category: Coding Practices [5]
- CWE subcategory: CWE-561 [2]

### Description

Since version 0.6.5, Solidity introduces the feature of declaring a state as immutable. An immutable state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as immutable is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an immutable state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of immutable states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the FWX LBPs protocol, we observe there are several variables that need not to be updated dynamically. They can be declared as immutable for gas efficiency.

```
contract Vault is ManagerTimelock {
    using SafeERC20 for IERC20;
    address public tokenAddress;
    ...
}
```

Listing 3.5: Vault

```
contract HelperBase is Manager {
    struct ActiveLoanInfo {
        uint256 id;
        uint256 currentLTV;
        uint256 liquidationLTV;
        uint256 apr;
        uint256 actualInterestOwed;
    }
    address public aphCoreAddress;
    uint256 public WEI_UNIT = 1 ether;
    uint256 public WEI_PERCENT_UNIT = 100 ether;
```

```
    }
```

Listing 3.6: HelperBase

**Recommendation** Revisit the state variable definition and make good use of immutable/constant states.

**Status** The issue has been addressed by the following commits: 0a99980 and 3798521.

## 3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

### Description

In the FWX LBPs protocol, there are a series of privileged accounts that play a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the accounts.

```
    function setPriceFeedAddress(address _address) external onlyAddressTimelockManager {
        address oldAddress = priceFeedAddress;
        priceFeedAddress = _address;

        emit SetPriceFeedAddress(msg.sender, oldAddress, _address);
    }

    function setCoreBorrowingAddress(address _address) external
        onlyAddressTimelockManager {
        address oldAddress = coreBorrowingAddress;
        coreBorrowingAddress = _address;

        emit SetCoreBorrowingAddress(msg.sender, oldAddress, _address);
    }
```

Listing 3.7: CoreSetting::setPriceFeedAddress()&&setCoreBorrowingAddress()

```
    function setPoolLendingAddress(address _address) external onlyAddressTimelockManager
         {
        address oldAddress = poolLendingAddress;
        poolLendingAddress = _address;

        emit SetPoolLendingAddress(msg.sender, oldAddress, _address);
    }
```

```
    function setPoolBorrowingAddress(address _address) external
        onlyAddressTimelockManager {
        address oldAddress = poolBorrowingAddress;
        poolBorrowingAddress = _address;

        emit SetPoolBorrowingAddress(msg.sender, oldAddress, _address);
    }
```

Listing 3.8: `PoolSetting::setPoolLendingAddress()&&setPoolBorrowingAddress()`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. The `multi-sig` mechanism could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

**Recommendation** Suggest to introduce the `multi-sig` mechanism to manage all the privileged accounts to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

**Status** The issue has been confirmed by the team. The team intends to introduce `timelock` mechanism to mitigate this issue.

## 3.7 Potential Protocol Risk from Low-Liquidity Assets

- ID: PVE-007
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `FWX LBPs` Protocol
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

With the occurrence of the `Mango Market Incident` on `Solana`, the risk of liquidity attacks on lending platforms attracts much attention from the entire `DeFi` community. In the following, we give one possible vector to illustrate the risk. A malicious actor may borrow a large amount of the available supply of a token (such as `ZRX`) from the lending market and sell it across multiple centralized and decentralized exchanges to depress the open market price. Once the price oracle of the lending market is updated with a lower price, the malicious actor may then withdraw most of the original collateral.

To elaborate, the malicious actor supplies `$30M` stablecoins as collateral firstly (Step I), secondly borrows `$20M` illiquid token (Step II), next sells it to depress the token's market price by `95%` and realizes `$7.5M` (Step III), and finally withdraws `$28M` collateral with the user's debt going down to `$1M` (Step IV). Overall, the malicious actor profits `$5.5M` leaving lending market with bad debt. The `Market Manipulation Risk` report [10] shows more details.

**Recommendation** Remove the low-liquidity assets from `FWX LBPs` to avoid the above risk of market manipulation.

**Status** The issue has been confirmed by the team. The low-liquidity assets will not be supported in the protocol.

# 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `FWX LBPs` protocol, which is a decentralized non-custodial liquidity markets protocol. The `FWX LBPs` protocol allows users to participate as depositors or borrowers. Depositors provide liquidity to the market to earn passive income, while borrowers are able to borrow in an over-collateralized fashion. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

# References


[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.

[10] Volt Protocol team. Market Manipulation Risk. https://github.com/volt-protocol/volt-protocol-core/blob/develop/audits/venue-audits/compound.md.